Lab 06: Probability and statistics, Tuning

This lab practices the maths we covered in lecture 6, including combinations and permutations. In the second part, you will use the secrets module to generate random numbers. Finally, you will expand the spell checking case study to compare the algorithms using statistics and graphs such as boxplots.

Part 1 Combinations, Permutations and probability

NOTE: There is a quiz in moodle for you to check some of the answers for the questions in this section

1. Students and birthdays

There are 42 students in a class.

a) How many different possibilities are there for choosing two class reps? (i.e. how many different pairs can be formed)
b) What are the chances that none of the students share a birthday with the teacher?
c) What are the chances that exactly one shares the teacher's birthday?
d) What is the probability that no pair share a birthday?

2. Playing cards

You have two packs of playing cards, a red one and a blue one. (52 cards per pack, 2 - 10, Jack, Queen, King, Ace in each of 4 suits)

Select one card from each deck.

a) How many possibilities are there for getting the same card from each pack?
b) What is the probability of getting two clubs?
c) What is the probability of getting two Queens?
d) What is the probability of getting two Aces of Spades?

3. 6-letter passwords

How many 6 letter passwords can be formed using the letters a-z?

a) If you are allowed to use each letter no more than once?
b) If each letter can be used multiple times?

4. Desert Island

Alan, Brian, Cecile, Dan and Emily are cast away on a desert island.

a) The first priority is to form a government of Prime Minister and Chancellor. How many ways are there to do this?
b) If they decide instead to form an executive committee with two "equal" members, how many ways are there to do this?

5. Communication

You have an unreliable communication stream. The probability that one bit will get flipped is p. N bits are transmitted, what is the probability that 0 bits get flipped?

6. Python generators

a) In the lecture, we looked at the script **permutations1.py**. Download this script from moodle and make sure you understand it. The new thing here is the use of "**yield**" which creates a **generator** rather than a list.

b) Read https://www.pythoncentral.io/python-generators-and-yield-keyword/ (make sure you look at the Python 3 example) to find out more about generators.

c) Another slide in the lecture showed a way of using Python to solve the problem of the original phrase that is represented by a given md5 hex digest. Adapt the script **permutations.py** to solve the following problem:
The cards of a 52-card deck are named like “King of Diamonds”, “Four of Hearts” etc.
bf460739f0400b3b5e0b10dfab528ea8 is the hex digest of the name of a card. Which one?
(see https://cdn.pixabay.com/photo/2013/07/13/13/47/card-deck-161536_960_720.png if you can't remember the different cards)

7. Script for calculating possible combinations/permutations, and listing them

The script **perm_com1.py** gives two examples for selection without repetition, one permutation and one combination.

To run the script, you need to make sure that scipy is installed. If not, use the command

```
C:\> py -3.7 -m pip install scipy
```

at the Windows command line.

Open the script, run it and inspect the code. Make sure you understand the script.

Modify the script to answer the following questions:

a) In a football penalty shootout, 5 different members of the team of 11 players are selected to participate. Calculate how many possible ways are there of doing this?

b) Assume the team of 5 has been selected, and consists of Jim, Alan, Gary, Sam and Owen. They now need to decide the order - who goes first etc.
   - Calculate how many different permutations there are
   - List all the permutations

8. Calculating Cumulative probability - cumulative.py

In the lecture, there were several examples related to cumulative probability - scoring at least 3 goals in penalty shootout, probability of no more than x machines failing.

Download **cumulative.py** from moodle. It is a working script that can be used to calculate these and other cumulative probabilities.

Use the script to answer the following questions.

a) In a penalty shootout, a team has 5 shots. Based on recent trials, the manager has estimated the success probability in each shot to be 85% (17 out of 20).
   - What is the probability that the team scores exactly 4 goals from the 5 attempts?
   - What is the probability that the team scores exactly 5 goals from the 5 attempts?
   - What is the probability that the team scores at least 3 goals?

b) We have 2000 machines. Each has a failure rate of 1:1000 per day.

- What is the probability that no machine fails in one day?
- What is the probability that exactly 1 machine fails in one day?
- What is the probability that up to 5 machines fail in one day?

9. How many spares?

cumulative.py contains the "stem" of a function called spares() to work out how many spares need to be stocked so that the probability of running out is less than "out".

Complete this function.

You will need to

- Work out the parameters that should be passed with each function call. This will have to be "out", plus two others. Add these to the brackets in the function definition.
- Use a while loop that calls cumulative() with appropriate parameters until the return value is more than "out".
- At this point, the function can return the number of spares required.

Test your function: From the lecture we know that

- We need 9 spares if we have 1000 machines with a probability of failure of 1/1000 each, and want the probability of running out to be less than 1:1000000
- We need 12 spares if we have 2000 machines with a probability of failure of 1/1000 each, and want the probability of running out to be less than 1:1000000

Does your function give the same results?

Finally use your function to answer the following question:

> You have 500 machines running continuously. The expected failure rate of one machine is 1:10,000 per day. How many spares do you need at the start of the day – to be sure that the chance of running out in that day is less than 1:1,000,000?

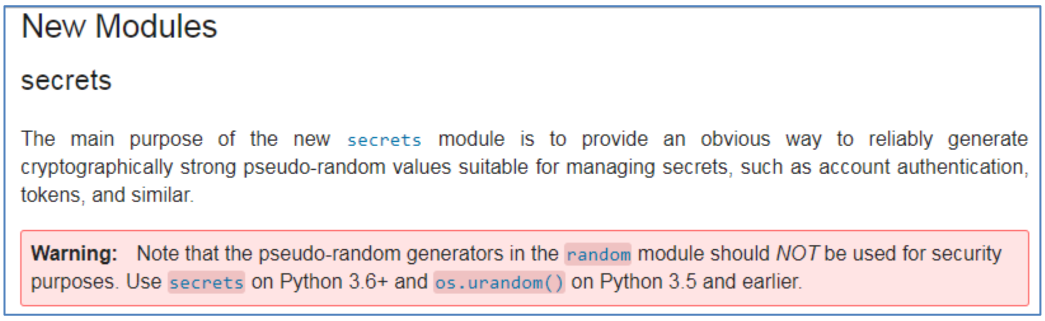
You can check your answer in the moodle quiz.

Part 2 the secrets module - random numbers

10. The secrets module

Many applications need random numbers generated. Where random numbers are used for security related purposes, they should be cryptographically secure. A new module, secrets, has been made available with Python 3.6 for this purpose. The old module, random, generates less secure pseudo-random numbers and should no longer be used.

According to the release notes at https://docs.python.org/3.6/whatsnew/3.6.html#new-features:



New Modules

secrets

The main purpose of the new `secrets` module is to provide an obvious way to reliably generate cryptographically strong pseudo-random values suitable for managing secrets, such as account authentication, tokens, and similar.

Warning: Note that the pseudo-random generators in the `random` module should *NOT* be used for security purposes. Use `secrets` on Python 3.6+ and `os.urandom()` on Python 3.5 and earlier.

Read the proposal for this module, the PEP, at https://www.python.org/dev/peps/pep-0506/. Focus on the Rationale and Frequently asked questions.

Q: Briefly summarise the rationale for introducing this module.

11. Random Numbers in Python

So let's use the new **secrets** module, in the Python standard library, for secure pseudo-random number generation. By default, random numbers are selected as a fraction between 0 and 1.

```
>>> import secrets
>>> secrets.randbelow(1000)
413
```

It is also easy to select random choices from ranges of numbers, using lists or list comprehensions.

```
>>> secrets.choice([i+1 for i in range(6)])
```

Execute this statement a few times.

Q: After how many executions do you get an answer that you had before?

Q: How many tries until you get the answer 1?

Q: What are the possible outcomes of this statement? Explain how this list comprehension works. Why is it using i+1 not i?

12. Random password generation challenge

The script shown below provides a skeleton for generating random passwords of a given length. Your tasks are to:

- Understand the script and add suitable comments that explain the code
- Fix the script
    - there is something missing that needs to be added to stop an error being generated
    - the return statement is missing
- Add suitable test cases to print three random passwords, 4, 6 and 10 characters long
- OPTIONAL extensions:
    - Modify the script so that the passwords can also contain the digits 0-9.
    - Ensure that passwords are printed as strings, for example, 'aB$7xJ', not printed as lists.
    - Replace the loop in the generate_password() function with a list comprehension.

You can copy & paste the code from below or download from moodle to create the initial script.

Script starting point:
random_password_start.py (in Moodle)

```
# Script:  random_password.py
# Desc:    generates random password using secrets module
# Author:  Petra L
# Created: 28/9/17
#
#

### something missing here
charset=list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")      # explain
charset.extend([x.lower() for x in charset])    # explain
charset.extend(list("@&%_$#"))                  # explain
### add digits 0-9 to charset too

def generate_password(length):
    """add suitable function doc string here"""
    password=[]
    for n in range(length):
        password.append(secrets.choice(charset))
    ### add suitable return statement

def main():
    ### test cases
    pass

# boilerplate that calls main() if run as script
if __name__ == '__main__':
    main()
```

Part 3 - Spellchecking case study

The following exercises continue the spell check case study from lab 4.

In case you haven't finished the script from lab 4, the downloads for Lab6 include **spellcheck_prepare.py** and **spellcheck_lab4.py**. The former is a solution for the pre-processing task. The latter implements the spellcheck using linear and binary search.

13. Setting up and implementing hashtable search

From lab 4, you should already have a file clean.txt which contains the pre-processed Frankenstein novel. If you are not sure about the quality of your pre-processing, or haven't got clean.txt yet, open **spellcheck_prepare.py** and run it.

Check that the file clean.txt has been created.

Next, open **spellcheck_lab4.py**. Inspect the code.

There is a third searching algorithm implemented, hash table search. Make sure you understand how this works. Lecture 5 has some slides that introduced this.

Run **spellcheck_lab4.py** and check that you are happy with it.

Q: What is the purpose of the line below?

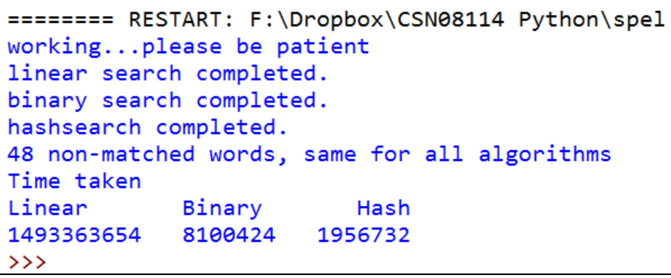
target = target[:1000]

14. Comparing the algorithms - timing your code

Adapt the script so that it times the execution of each of the algorithms and outputs the times taken.

Use **time.perf_counter_ns()** to calculate the time taken. Refer to lecture and lab 5 for examples of how to use this.

The output at this stage should look something like this:



```
======== RESTART: F:\Dropbox\CSN08114 Python\spel
working...please be patient
linear search completed.
binary search completed.
hashsearch completed.
48 non-matched words, same for all algorithms
Time taken
Linear       Binary       Hash
1493363654   8100424   1956732
>>>
```

Q: Which is the fastest algorithm? Which is the slowest?

Q: Compare this with the theoretical expectations. Are there any surprises?

Q: Run the script two more times, without any modifications. What do you notice about the results?

Q: What could be the reason for these variations?

15. Comparing the algorithms: Boxplots

Because there is a lot of variability, we should run each algorithm several times and store the results of all the separate runs for later processing.

To do this, modify your script to wrap the timing code into a loop which runs 10 times. The results must be written to three separate lists, one for each of the algorithms.

During development, change the line

```
target = target[:1000]
```

to

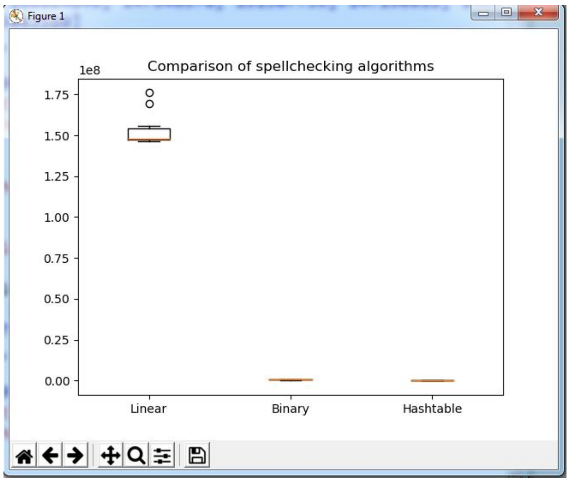
```
target = target[:100]
```

so that your code runs faster!

For example:

```
==== RESTART: F:/Dropbox/CSN08114 Python/spellCheck/spellcheck_boxplot.py ====
linear = [217202700, 177960592, 178757834, 146533638, 155615145, 146182863, 1476
22791, 147587472, 147023273, 153761955]
binary=[1618334, 606762, 489334, 532804, 483297, 541256, 540652, 481787, 531898,
 481485]
hashs=[43771, 32602, 32904, 31394, 31093, 32904, 32300, 32300, 31697, 31998]
```

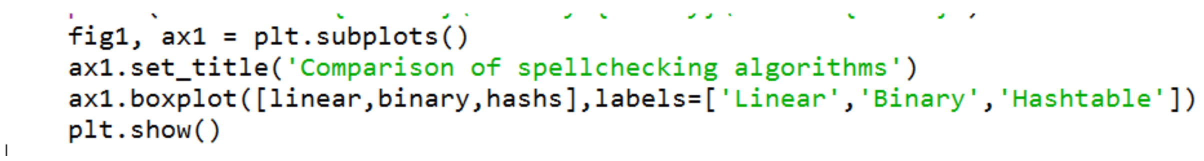
Once you have this working, we can create summaries and graphs of the data to show the results more clearly. Add code to your script to create a boxplot of the data, which looks like this:



You will need to import

```
import matplotlib.pyplot as plt
```

and then code similar to this:



```
fig1, ax1 = plt.subplots()
ax1.set_title('Comparison of spellchecking algorithms')
ax1.boxplot([linear,binary,hashs],labels=['Linear','Binary','Hashtable'])
plt.show()
```

The plot shows clearly that linear search takes much longer than the other two, but we can't clearly see the differences between binary and hashtable search algorithms.

16. Comparing the algorithms: mean and standard deviation

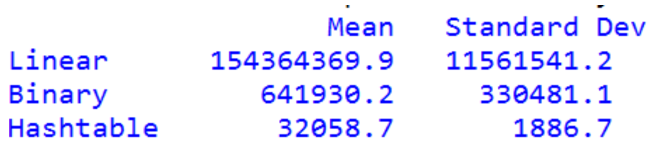
Since Python 3.4 there is a statistics module in the standard library, which has methods mean and stdev for calculating the mean and standard deviation respectively.

Try the following code

```
>>> hashs=[35319,32603,32300,35922,34112,35017,32300,32300,32602,34111]
>>> import statistics
>>> statistics.mean(hashs)
>>> statistics.stdev(hashs)
```

Does this code work? Do you understand it?

Then you are ready to add code to your script to calculate the mean and standard deviation for the three algorithms. Your output should look like this



```
                      Mean    Standard Dev
Linear         154364369.9    11561541.2
Binary            641930.2      330481.1
Hashtable          32058.7        1886.7
```

17. Tuning the code

We know that the hashtable search is by far the fastest. So, if you were to build a spell checker for commercial use, you would use this algorithm.

Q: What else could you do to make the spell checker perform faster?

This is an open question and asks for ideas. For feedback, discuss your ideas with a tutor in a lab. You are not required to implement and test your ideas.

18. **OPTIONAL EXERCISE**: Big-O of the three algorithms

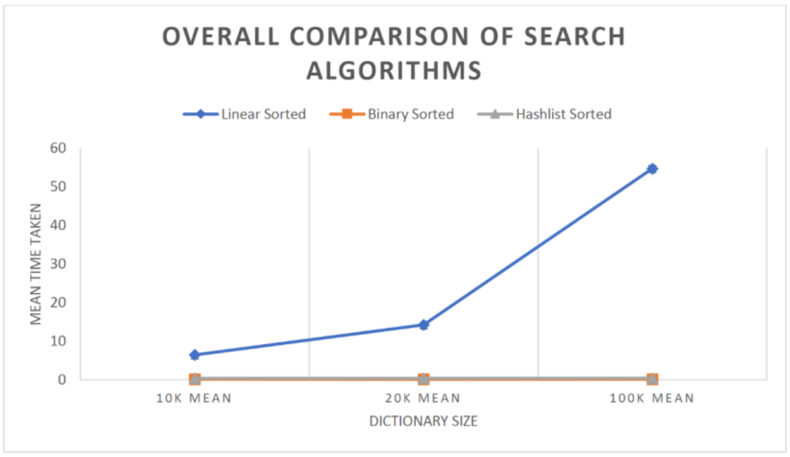
In lab 4, you were given links to three "dictionaries" of different size. By running our spellchecking script with each of the three dictionaries, we can measure how the runtime increases with increasing input. This is what we need to estimate the big-O of the algorithms and visualise them.

Adapt your script, or write a separate "wrapper" script, to run each algorithm 10 times for each of the three different size dictionaries. As before, use just the first 1000 words from clean.txt for the checking.

To summarise the results, output the mean running time for each algorithm and each word list in a table of this format:

	Linear search	binary search	hashtable search
10k dictionary			
20k dictionary			
100k dictionary			

Once you have all these results, you can finally plot them on a graph laid out like the one shown on the right. Refer to lecture / lab 5 for help with plots.



The theoretical expectations are that linear search should be O(n). If this is the case, the 20k dictionary should take about twice as long as the 10k dictionary, and the 100k dictionary about 5 times as long as the 20k one or 10 times as long as the 10k one.

Q: Is this the case in your tests?

Q: Binary search should be O(log(n)), so the 100k dictionary should take approx. twice as long as the 10k one (because 10^2=100). Is this the case?

Q: Finally, hashtable search should be O(1), so all searches should take about the same time. Is this the case?